Problem A. 6789

Problem idea : Seunghyun Joe (ainta)
Problem preparation : Seunghyun Joe (ainta)
First solver: Polish Mafia : Sokolowski, Radecki, Smulewicz (00:04)
Total solved team: 119

The whole grid is point symmetric if for every cell (i, j) $(1 \leq i \leq N, 1 \leq j \leq M)$, the opposite cell $(N + 1 - 1, M + 1 - j)$ is in a point symmetric shape. Let's pair those cells with opposite side cells. There are two cases:

- Two paired cells are the same. Then it should always be 8. Otherwise, we can't make a magic matrix. This case can occur in the center if N, M are both odd.
- Two paired cells are different. If both cells are 8, then they are already symmetric. If both cells are 7, then you should rotate one of them. If both cells are 6 or 9, then you should rotate one of them if and only if they are equal. Otherwise, we can't make a magic matrix.

Shortest solution: 381 bytes

Problem B. Bigger Sokoban 40k

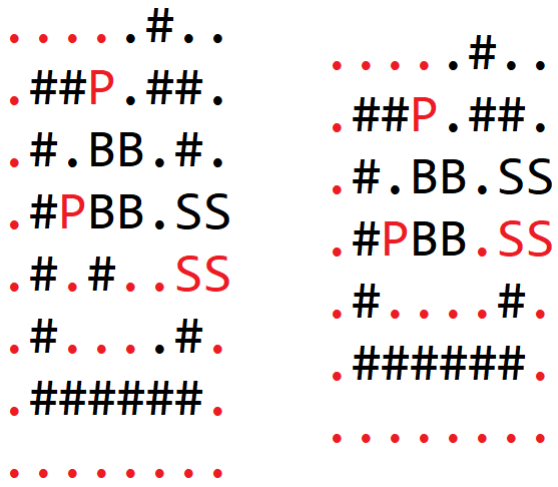
Problem idea : Gyeonggeun Kim (kriii)
Problem preparation : Gyeonggeun Kim (kriii)
First solver: SPb ITMO University 1 : Sayutin, Kirillov, Budin (03:18)
Total solved team: 3

If we think about how large the minimum number of moves can grow, there is an upper bound of $O((NM)^2)$. This is because the problem of finding the minimum can be solved by a breadth-first search, with each state (x_s, y_s, x_p, y_p) denoting that the upper-left part of the box is in (x_s, y_s), and the player is in (x_p, y_p). But can we really construct a grid that requires $\Omega((NM)^2)$ moves?

To achieve this, we must force the player to use at least a constant fraction of states. This means two things: first, the box must be pushed all the way around the grid. Second, after pushing the box, the player must go all the way around the grid in order to push again.

For the first condition, just build a long twisted corridor. For the second condition, we will create the following situation: we push the box, then we want to push it in a different direction, but we must go all the way around to reach the other side of the box.

There are many ways to build such a corridor. For example, consider grids like following:



We can connect these turning points to make a bigger grid. Turning the direction of the box in each structure will force the player to tour the entire part of the grid, as specified above in the red characters.

In a 49×51 grid, we can put at least $2500/5^2 - 2 \cdot (50/5) = 80$ turning points somehow, and each touring will need at least $2500/5 = 500$ moves. So we can make the grid that needs at least 40k moves to solve. You can either write a code to generate the grid, or build the grid by hand and hardcode it into the solution. In any case, it's highly recommended to write a checker that counts the required number of moves.

Shortest solution: 2568 bytes

Problem C. Cleaning

Problem idea : Geunwoo Bae (functionx)
Problem preparation : Geunwoo Bae (functionx)
First solver: Kazan+SPb : Rakhmatullin, Gainullin (04:36)
Total solved team: 2

Since the problem is about the paths on a directed graph, group the vertices by SCC first. Each SCC is a connected component on the grid.

When you add SCCs by topological order (for A $\rightarrow$ B, A is added earlier), the shape is a group of rectangles that do not intersect in the grid. It is because the directions of the boundary grids outside SCCs are fixed. For each rectangle, the following lemma is satisfied.

Lemma 1. *For each rectangle, at least one of the following two sentences is true:*

1. *For every cell inside a rectangle, the path going past the left boundary and the right boundary both exists.*
2. *For every cell inside a rectangle, the path going past the upper boundary and the lower boundary both exists.*

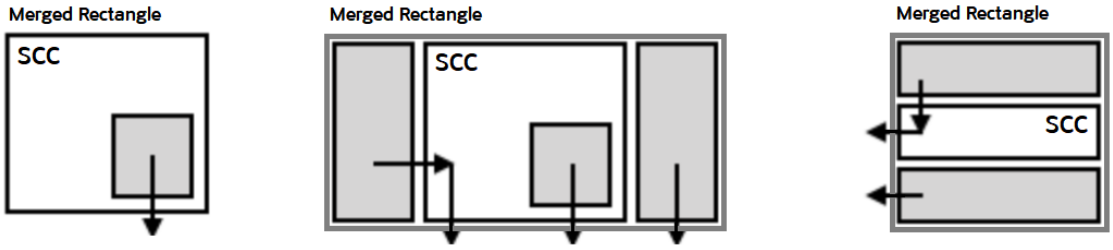
Proof. Let's use mathematical induction.

- Consider the SCC does not intersect with other SCCs that have higher priorities. If the upper boundary is unreachable, all of the directions of the upmost grids are `U`. Since the rectangle is SCC, both upper left position and upper right position is reachable. You can reach the left boundary and the right boundary by going to upmost position and getting out the rectangle. Similarly, you can reach the left boundary and the right boundary when the lower boundary is unreachable.

- For other cases, intersections with other SCCs always appears as one of three types at the following figure. For the first type, the merged rectangle is the smallest rectangular region that covers the SCC. Thus, for all four directions, there are at least one reachable grid on the SCC. Assume that there are no path going past the upper boundary. Then all the directions of the upmost grids on the SCC are `U`. If there is a smaller rectangle located at the upper boundary, the path from the upmost grid on the SCC to the upmost grid inside the smaller rectangle exists because it is free to move left or right. It contradicts that the SCCs are added by topological order. Thus, there are no smaller rectangles. It means that all of the directions of the upmost cells are `U`, and you can reach the left boundary and the right boundary. Similarly, you can reach the left boundary and the right boundary when the lower boundary is unreachable. As a result, both upper and lower boundary are reachable if either left or right boundary is unreachable. If you start in the smaller rectangle, the path $(SmallRect) \to (SCC) \to (Outside)$ exists.

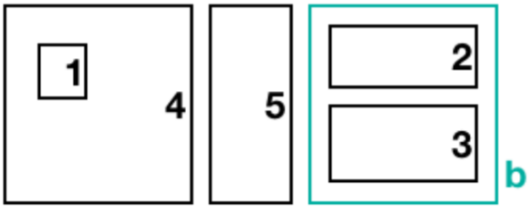
  For the second type, all of the directions of the leftmost grids in the smallest rectangular region that covers the SCC are `L` (only if there is a rectangle at left). You can reach the upper boundary and the lower boundary by using the leftmost grids (if there are no rectangle at left, you can use the rightmost grids instead). If you start on the rectangle at the leftmost or the rightmost position of the SCC, either the path $(Rect) \to (Outside)$ or the path $(Rect) \to (SCC) \to (Outside)$ exists.

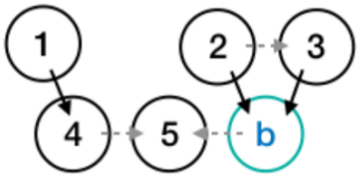
  For the third type, the proof is similar to the second type.



□

To solve the problem, we have to model the graph in $O(NM\alpha(NM))$ and solve each query in $O(log(NM))$. Each node of the graph represents either a SCC or a virtual node. Each node covers a rectangular region on the grid. Each edge represents a *tree edge*, a directed edge from a node to its parent, or a *sibling edge*, a directed edge from a node to its sibling. If all the tree edges are removed, for each node, either in-degree or out-degree is less than or equal to 1. A query on the model is given as a path on the tree.





To construct the model, add SCCs in topological order and manage the rectangles. For each rectangle, store its position and the escape directions mentioned in the lemma.

When each SCC is added, the edges are added by the following instructions, which can be implemented by Union-Find Data Structure.

1. For every rectangle inside the smallest rectangular region that covers the SCC, add the tree edges to the SCC. Merge all rectangles and SCC into a rectangle R.
2. For all rectangles at the leftmost position of R, merge the rectangles, add a virtual node, and add the tree edges from the rectangles to the virtual node.
3. Repeat this for all rectangles at the rightmost, upper, lower positions of R.
4. Add the sibling edges from the virtual nodes to R. Some edges are not added for some situations.

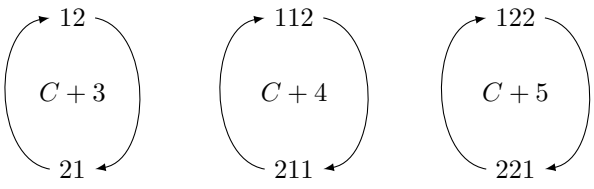
Author's Note: Since the depth of tree edges does not exceed $N + M$, there is a $O(Q(N + M))$ solution. The problem would have been better if the constraints were $NM \leq 10^6$ rather than $N, M \leq 10^3$.

Shortest solution: 5316 bytes

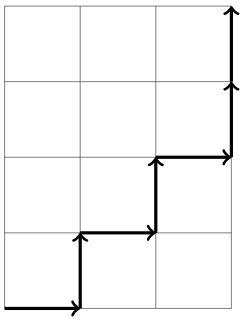
Problem D. Container

Problem idea : Gyeonggeun Kim (kriii)
Problem preparation : Gyeonggeun Kim (kriii)
First solver: Intellectual + AESC : Savkin, Lifar, Shekhovtsov (03:32)
Total solved team: 2

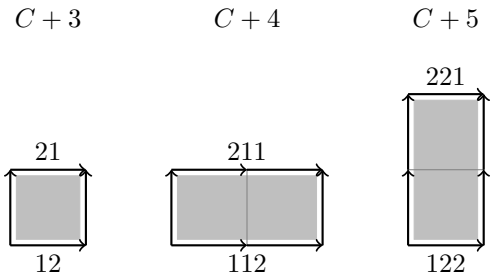
Given a binary string, this problem allows three operations about substring manipulation as follows:



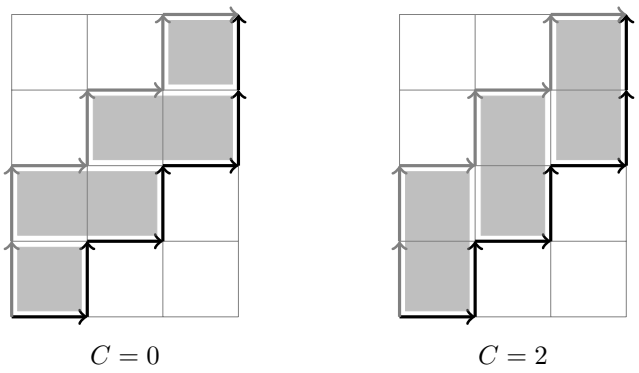
Now, we present the binary string as a path on the grid. `1` means move 1 toward the x-axis, `2` means move 1 toward the y-axis. The exmaple about `1212122` follows:



Then we can represent the three operations as the tiles of size 1×1(monomino) or 1×2(domino).



To make the target string in optimal cost, you should find an optimal tiling of monominoes and dominoes between the path of the given string and the path of the target string. The example when the given string is `1212122` and the target string is `2212121` follows:



When we set the cost of every string as a minimized $(C+3) \cdot \#(\square) + (C+4) \cdot \#(\square\square) + (C+5) \cdot \#(\begin{smallmatrix}\square\\\square\end{smallmatrix})$, it can be proved every transition is relaxed. Proving this is messy, so I omit it.

Getting optimal tiling is done with the min-cost flow. Basically, put a $\square$ in every cell.
When substituting two $\square$s to a $\square\square$, it costs $(C+4) - 2 \cdot (C+3) = -(C+2)$.
When substituting two $\square$s to a $\begin{smallmatrix}\square\\\square\end{smallmatrix}$, it costs $(C+5) - 2 \cdot (C+3) = -(C+1)$.

The grid is modeled as the bipartite graph, so you can use the min-cost flow and solve this in $O(N^4 \lg N)$ time complexity. But this seems slow.

The length of the successive shortest path in the min-cost flow is monotonically increasing, so making pre-flow for every possible [redacted] doesn't affect the optimality of the solution. Then processing the rest $O(N)$ flow can be done in $O(N^3 \lg N)$ time.

The actual order of the process can be arranged with the topological sort.

Note that the shortest solution is implemented with alternative DP approach that runs in $O(N^2)$ time. Contact `ainta` for more details.

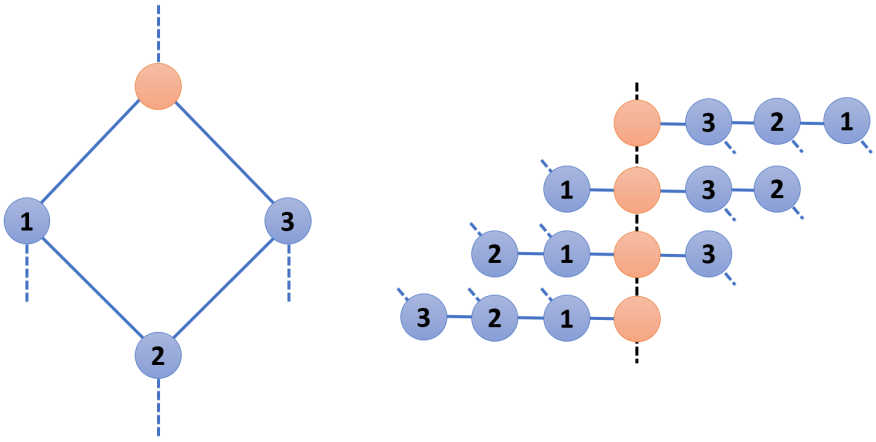
Shortest solution: 2359 bytes

Problem E. Dead Cacti Society

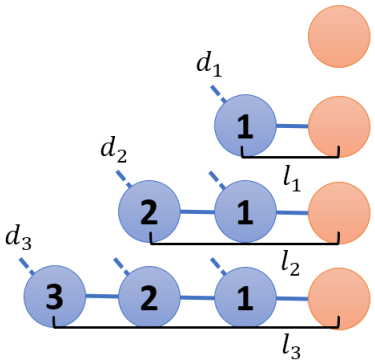
Problem idea : Gyeonggeun Kim (kriii)
Problem preparation : Gyeonggeun Kim (kriii)
First solver: Polish Mafia : Sokolowski, Radecki, Smulewicz (02:45)
Total solved team: 2

First, decompose the cactus into cycles and edges. Now, you can determine whether there exist some processed trees whose diameter is not greater than R in $O(N + M)$ time.

To process the decomposed cactus, in reverse DFS order, for each cycle and edge calculate the minimum possible farthest distance from the pivot vertex where the diameter has not exceeded R. For example, there is a cycle of size 4, the orange vertex is the pivot vertex in the following figure. There are 4 possibilities to make the tree.



We process the left half and right half separately. For the left half, let l_i be the distance from the vertex i, and d_i be the minimum possible farthest distance from the vertex i when vertex i is the pivot. For convenience, the pivot vertex is the vertex 0.



First, the process for the left half. Let R_i be the diameter when we consider vertex 0 to vertex i, D_i be the minimum possible farthest distance from the pivot vertex when we consider vertex 0 to vertex i. Then $R_i = \max_{0 \le p < q \le i} [(d_p - l_p) + (d_q + l_q)]$ and $D_i = \max_{0 \le p \le i}(d_p + l_p)$. We can make recurrence for R and D.

First, $D_i = \max(D_{i-1},\ d_i + l_i)$. And we define $X_i = \max_{0 \le p \le i}(d_p - l_p) = \max(X_{i-1},\ d_i - l_i)$ for convenience. Then, $R_i = \max\big(R_{i-1},\ X_{i-1} + (d_i + l_i)\big)$. So, we can get all the R, D values incrementally.

The process for the right half is the same as above too. Similarly, we can combine the left half and the right half.

We can use a binary search in R. So, we can solve this problem in $O\big(N \lg(N \cdot X)\big)$ time where X is the maximum length of edges.

In the actual problem, the new edges are regenerated from cutting side. Generalization is straightforward, so I omit it.

Shortest solution: 2945 bytes

Problem F. Hilbert's Hotel

Problem idea : Jaemin Choi (jh05013)
Problem preparation : Jaemin Choi (jh05013)
First solver: Past Glory : Borys Minaiev, Gennady Korotkevich (00:59)
Total solved team: 60

This problem has two independent components: the "2 g x" queries and the "3 x" queries.

2 g x

The x-th number of the group g can be expressed as $a_g x + b_g$ where a_g and b_g are constants depending on g. Initially, $a_0 = 1$ and $b_0 = 10^9 + 6$ (because we are taking modulo $10^9 + 7$). When k people arrive at the hotel, we increase every b_i by k, then set $a_G = 1$ and $b_G = 10^9 + 6$. When infinitely many people arrive, we double every a_i and b_i, then set $a_G = 2$ and $b_G = 10^9 + 6$.

This takes $O(Q)$ per query. Now let's step back and look at the big picture. After n groups arrive, we have $2n+1$ linear functions $U_1, U_2, \cdots, U_n, S_0, S_1, \cdots, S_n$; each U_i represents the updates to the previous groups when group i arrives, and each S_i represents the initial state of group i. Given a group number g, we want to evaluate $G_g = U_n U_{n-1} \cdots U_{g+1} S_g$, where $f_1 f_2$ denotes the composition of two functions f_1 and f_2.

Denote $H_0(x) = x$ and $H_n = U_n U_{n-1} \cdots U_1$, then $H_{n+1} = U_{n+1} H_n$ and $G_g = H_n H_g^{-1} S_g$. The inverse of a linear function $f(x) = ax + b$ is $f^{-1}(x) = a^{-1}x - a^{-1}b$. By computing a^{-1} using Fermat's little theorem, you can answer the query by evaluating only three linear functions.

An alternative solution is to use a segment tree with lazy propagation. Each node has a pair $\{a, b\}$, meaning either "I will apply $ax + b$ to all of my child nodes" or "I am the leaf node, and my pair corresponds to $\{a_g, b_g\}$." The details will be straightforward if you are familiar with this kind of data structure.

3 x

We trace back the history of the room numbers occupied by the guest currently in room x. Process the "1 k" queries received so far backwards, keeping track of the value x:

- If $k > 0$: if $x \geq k$, then decrease x by k. Otherwise, we found the group number.
- If $k = 0$: if x is even, then divide it by 2. Otherwise, we found the group number.

This can lead to $O(x + Q)$ per query in two different ways. The first case is when the "1 k" queries are full of $k > 0$, but x is too large and k's are too small. The second case is when the "1 k" queries are full of $k = 0$, but x is zero. That means we don't want to keep subtracting, and we don't want to keep dividing if $x = 0$.

We group the consecutive "finite people" queries and the consecutive "infinite people" queries. Let's call them the "blocks." Process the blocks backwards:

- Finite people block: If the number of guests in this block is $\leq x$, then decrease x by that number and skip this block. Otherwise, our guest came from this block. Save the prefix sums for each block, and apply binary search to find the group number.
- Infinite people block: If $x = 0$, then skip this block. Otherwise, keep dividing by 2 until it becomes odd or this block is used up.

Every time you skip two blocks, x decreases by at least half. Therefore this process stops after $O(\log x)$ blocks, and the time complexity reduces to $O(\log x + \log Q)$ per query.

Shortest solution: 1507 bytes

Problem G. Lexicographically Minimum Walk

Problem idea : Jaehyun Koo (koosaga)
Problem preparation : Suchan Park (tncks0121)

First solver: USA1 : Kevin Sun, Scott Wu, Andrew He (00:12)
Total solved team: 113

We will assume that there is a path from $S \to T$, which can be determined with DFS. By definition, you should minimize the label of first edge, and then second edge, and then third edge, and so on. Thus, a greedy strategy that only considers the best edge in every moment is sufficient to solve the problem: You simply pick the edge that have the minimum label, move toward the edge, and repeat.

Let's be more precise. The greedy algorithm should first guarantee that the edge is **feasible** (i.e. you can take the edge and reach the vertex T in 10^{100}), and among them it's minimum. The first condition can be checked by a depth-first search: Reverse the direction of the input graph, and find the set of vertices that are reachable from T. Since we reversed the direction of graph, these vertices are exactly the one which can reach the vertex T. If we can reach the vertex T, then it could be reached in at most N steps, so we don't have to care about its distance unless we used over $10^{100} - N$ steps. Thus, the feasible condition can be easily determined in $O(N + M)$ time, and you can ignore all edges that leads to an infeasible vertex.

After guaranteeing the feasibility, the greedy algorithm should find the **minimum** label edge. If you are already at the vertex T, obviously you should stop. Otherwise, you can simply search for the feasible edges with minimal label, and go toward that direction.

Last but not least issue is the time complexity. If you simply search for the edge with minimum label and repeat it for 10^6 times, you will result in the time complexity of something like $O(10^6 \times N)$. This is too slow. However, consider the case where you visit the node that you've visited before: After visiting that node, you know that you will cycle that node almost (little bit less than 10^{100} step) forever, and you will never reach T in 10^6 step anyway. So, if you found the node that had been visited before, just print `TOO LONG` and terminate. Since you only visited each node at most 1 time, and each edge is considered at most 1 time, the time complexity here is $O(N + M)$.

Shortest solution: 885 bytes

Problem H. Maximizer

Problem idea : Gyeonggeun Kim (kriii)
Problem preparation : Gyeonggeun Kim (kriii)
First solver: Polish Mafia : Sokolowski, Radecki, Smulewicz (00:11)
Total solved team: 109

Exactly when is the difference sum maximized? Without the loss of generality, assume B is sorted in descending order. Then when we sort A in ascending order, we can get the maximum. This can be proven by the exchange argument. But, there is another way to achieve the maximum.

Let's first assume N is even, so $N = 2K$. After sorting A and B, the integers in $[1, K]$ are matched with the integers in $[K + 1, 2K]$. The integers in $[K + 1, 2K]$ are always bigger than the integers in $[1, K]$, so shuffling

the integers in $[1, K]$ and shuffling the integers in $[K+1, 2K]$ shouldn't be changing the sum of differences. Thus, the answer is the minimum number of swaps to match the integers in $[1, K]$ from A and the integers in $[K+1, 2K]$ from B in the same index.

Next, let's assume N is odd, so $N = 2K-1$. Similar to even, but there is K in this case. K can match with either the integers in $[1, K-1]$ or the integers in $[K+1, 2K-1]$, which needs more case handling. So, there are two cases: match the integers in $[1, K-1]$ from A and the integers in $[K+1, 2K-1]$ from B, or match the integers in $[1, K]$ from A and the integers in $[K, 2K-1]$ from B.

To find the minimum number of swaps to match a pair of intervals, Look at the ascending indices of the chosen integers, and compute the sum of the differences of the matching indices. Thus, we can solve every case in $O(N)$ time complexity.

Shortest solution: 473 bytes

Problem I. Minimum Diameter Spanning Tree

Problem idea: Jaehyun Koo, Suchan Park (koosaga, tncks0121)
Problem preparation : Suchan Park (tncks0121)
First solver: Kazan+SPb : Rakhmatullin, Gainullin (00:21)
Total solved team: 36

Let's think about the structure of the diameter in a tree. Consider the midpoint of a diameter, which is commonly known as the **center** of the tree. The center may not be a vertex and lie inside an edge. Given the diameter of the tree, it's trivial to find the center of tree. Interestingly, it turns out that we can also do the opposite: Suppose that we want to find the minimum diameter spanning tree given the center of the tree. Then we can simply take the **shortest path tree** starting from the center, and call it as an answer.

This is true because every diameter of the tree passes the center. If we know where the center is, then the length of a diameter is the sum of the distance for the farthest node, and the second-farthest node. In fact, if it's really a center of the tree, then the second-farthest node has the same distance as the farthest node. Thus, the strategy of minimizing the distance for each node is optimal.

Now, we will try to brute force the center of tree. Let's just fix the edge $e = \{u, v, W_{u,v}\}$ where the center belongs. We can see that the center will have two children u, v, and other $N-2$ nodes will be in the subtree of u or v. Fix two vertices far_u, far_v, which is the farthest vertex that is in the subtree of u, v, respectively. If we fix these vertices, the center can be found. If the center is in the edge as we anticipated, we can run a shortest path algorithm from this center and find the actual answer in $O(n^2)$ time. Combined with the number of candidates $O(n^4)$, we have an $O(n^6)$ poly-time algorithm.

Let's optimize this. First, it will be good to avoid using a shortest path algorithm for each center. The length of a diameter is obvious, so if we can be sure if it's a valid candidate, we can just take the minimum. Consider all $N-2$ other vertices x. By our assumption, all $N-2$ other vertices x should have either

$dist(x, u) \leq dist(far_u, u)$ or $dist(x, v) \leq dist(far_v, v)$. And if these conditions are all satisfied, then the shortest path algorithm will report that far_u, far_v are actually the farthest. So we don't need to run a shortest path, and the complexity is $O(n^4)$.

To reduce to $O(n^3)$, Let's just fix the far_u. Then, far_v should be the farthest vertex from v that is not covered by u. So fixing the far_u automatically fixes far_v. Such far_v can be found in $O(1)$ amortized time, if we precalculate the sorted array of distances for each v in $O(n^2 \log n)$. One might be worried that fixing the far_v that way will set the center outside of the edge. This concern is actually true, but in that case the optimal answer will lie in that direction of the center, so we can simply ignore such case and still enumerate all the candidate.

Shortest solution: 2413 bytes

Problem J. Parklife

Problem idea : Jaehyun Koo (koosaga)
Problem preparation : Jaehyun Koo (koosaga)
First solver: U of Tokyo UT a.k.a ls : Inoue, Isa, Takaya (00:25)
Total solved team: 61

The bridge connecting the point S_i, E_i is visible in the interval $[S_i, E_i)$. Since no bridges are crossing, for any pair of distinct bridge, their visible intervals are either disjoint, or contained in another. Since the intervals are disjoint or nested, we can observe a characteristic that is very similar to the *bracket sequence*. For a bracket sequence, every opening bracket has a matching closing bracket, and no pairs of brackets intersect. With this analogy we can build a following reduction: Given a bracket sequence, find a maximum subset of paired brackets which has at most K brackets nested.

A bracket sequence yields a parse tree, so let's solve the problem on the parse tree. Matching brackets form the vertices, and the ones directly contained in the sequence form the edges. In this parse tree, we want to find a maximum subset of vertices in which every path to the root contains at most K selected vertices.

Now we know this problem is somehow related to tree, so before moving on let's see how to build a tree. With the interval representation, although we can't know the tree, we can know the *preorder sequence* of the tree: If we sort the edges in an increasing order of S_i (and in case of tie, decreasing order of E_i) we exactly get the preorder sequence. From the preorder sequence, we can recover the tree using stacks. You can simply go over the preorder sequence, and maintain the vertices that are in the path to the root. Since you can know whether an interval is an *ancestor* to another interval (by checking the containment), this part can be simply done in $O(N)$ time excluding the sorting. Note that, this will actually yield a forest instead of a tree, so it will be helpful to just make the auxiliary root (for example, an interval $[-10^7, 10^7]$ with cost 0).

Now, we will try to solve the problem on the tree with dynamic programming. Let $DP_{i,j} = \{$maximum answer for the subtree of i where every path has at most j selected vertices.$\}$. For computing $DP_{i,j}$, if you select node i, then you can pick at most $j - 1$ nested nodes for each subtree. Otherwise you can freely pick

j nested nodes. This yields a simple $O(n^2)$ tree DP which is quite slow. By observing that j doesn't have to be larger than the maximum depth in subtree i, we can have an $O(n \log n)$ algorithm on balanced trees, but this is not a case for all inputs.

Now we will prove the following fact:

Lemma 2. *For any i, $DP_{i,*}$ is upward convex.*

Proof. We use induction. For leaf nodes this is obvious. For non-leaf nodes, Let's only consider the case of not selecting node i. By inductive hypothesis, you only add a constant or add two upward convex functions, which will again yield an upward convex function. If we add new node i, we are taking $g(i) = max(f(i-1)+w, f(i))$ to some upward convex function f. This is a Minkowski sum of f and a vector $(1, w)$, which is again convex. □

Note that the calculation of $g(i) = max(f(i-1)+w, f(i))$ is just a Minkowski sum: If you store the derivative in a sorted set, then taking such function is simply adding a value w to the sorted set. Thus, we can maintain such the set with heap: For leaf nodes, the heap only contains the single value w. For non-leaf nodes, we merge all heaps (by adding the values pairwise) and insert a single value w. Merging two heaps can be simply done in the size of the smaller heap, which is enough to yield an $O(n \log^2 n)$ time complexity by the argument used in Union-Find algorithm. (This argument is colloquially known as *small-to-large method*, so you can google for relevant information.)

However, I know you are not satisfied with $O(n \log^2 n)$ algorithm in such a small time limit, so here is the bonus:

Lemma 3. *The above algorithm runs in $O(n \log n)$ time.*

Proof. The above algorithm inserts at most n values into the priority queues. If an element is removed from a priority queue, it is merged with other values from another priority queue, and never appears again. So every value is inserted and deleted for at most $O(n)$ times. □

Although the logic is pretty involved, the actual code is quite short. (Tester `kdh9949`'s code is less than 60 lines)

Shortest solution: 1314 bytes

Problem K. Wind of Change

Problem idea : Jaehyun Koo (koosaga)
Problem preparation : Jaehyun Koo (koosaga)
First solver: Polish Mafia : Sokolowski, Radecki, Smulewicz (01:02)
Total solved team: 8

Construct a centroid decomposition for both trees T_1, T_2. Say that two nodes i, j have a lowest common ancestor L_1, L_2 in each centroid, then the distance between these two node is $dist(T_1, L_1, i) + dist(T_1, L_1, j) + dist(T_2, L_2, i) + dist(T_2, L_2, j)$. Since every node has at most $\log N$ ancestors on the centroid decomposition,

for a fixed i there exists at most $\log^2 N$ pairs of ancestors, so we may try to iterate all L_1, L_2 for each i.

Now, we have to solve a following problem: Given a node L_1, L_2, find a node $j \neq i$ with smallest value of $dist(T_1, L_1, j) + dist(T_2, L_2, j)$, which has a pair of LCA with i on **exactly** node (L_1, L_2). Having an **exact** condition makes life harder, so let's just relax this condition, and say that we need to find a node $j \neq i$ which is in the subtrees of both nodes (L_1, L_2). In this case, we can just enumerate all $N \log^2 N$ tuples of (ancestor on tree 1, ancestor on tree 2, $dist(T_1, L_1, x) + dist(T_2, L_2, x)$), and for each first / second value, maintain the first and second minimum of the third value. By carefully avoiding sort or hashmaps (simple DFS on centroid decomposition suffices), this part can be done in $O(N \log^2 N)$ time and $O(N \log N)$ memory.

Now I'll show that the above solution is actually just correct. We can see that the solution actually considers all possible pairs, so it never gets longer than the answer. Also, since $dist(T_1, L_1, i) + dist(T_1, L_1, j) \geq dist(T_1, i, j)$, it never gets shorter.

There exists an alternative solution that uses the same relaxation idea, but considers a centroid decomposition for only T_1. In this case, we have to minimize $dist(T_1, L_1, i) + dist(T_1, L_1, j) + dist(T_2, i, j)$. We can do this by finding a compacted tree, and solve this problem with DP. (To learn about compacted trees, you can try solving "Factories" from JOI Open 2014.) This solution is less beautiful and more technical, but there are two good things: This solution can be optimized to $O(N \log N)$ time, and you can compute the MST on the complete graph with weight $W(i, j) = dist(T_1, i, j) + dist(T_2, i, j)$ in $O(N \log^2 N)$ with this solution and the $O(N)$ solution from "Tree MST" in AtCoder. With another method you can reduce the complexity to $O(N \log N)$, which means that we just found the funniest way to compute the Manhattan distance MST in $O(N \log N)$.

Shortest solution: 2062 bytes